Decider

KRUG 27.10.2025

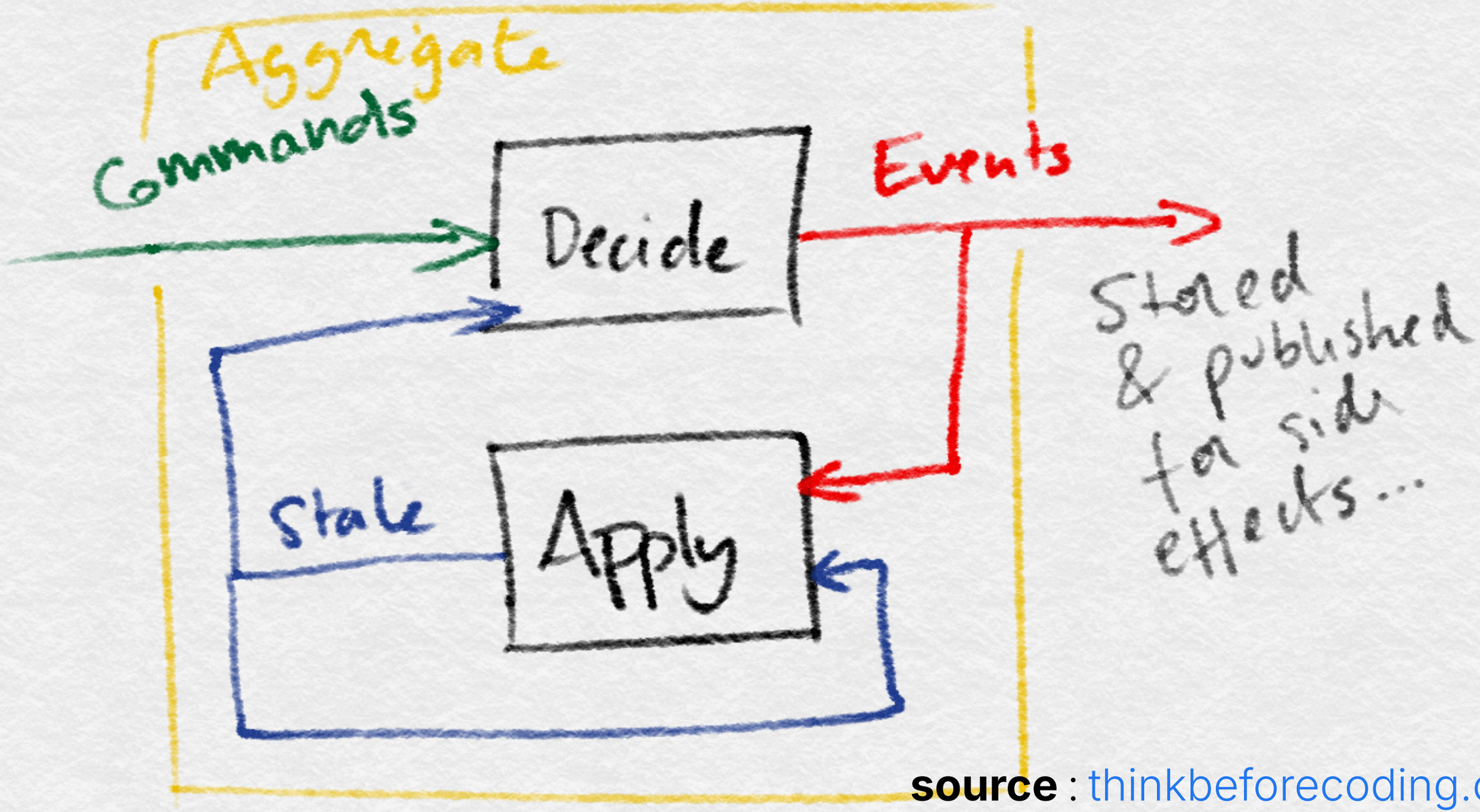
I’m Jan

 **DRUG** 

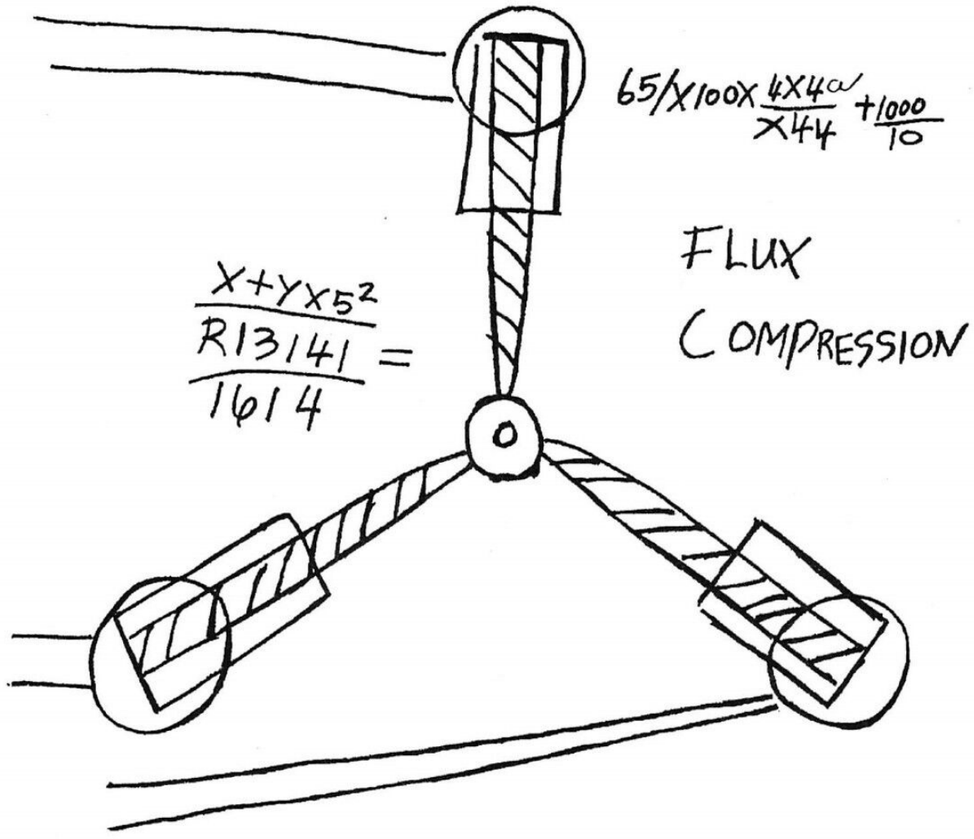
Every 3rd Monday of the month

17-19th April 2026

papercall.io/wrocloverb2026



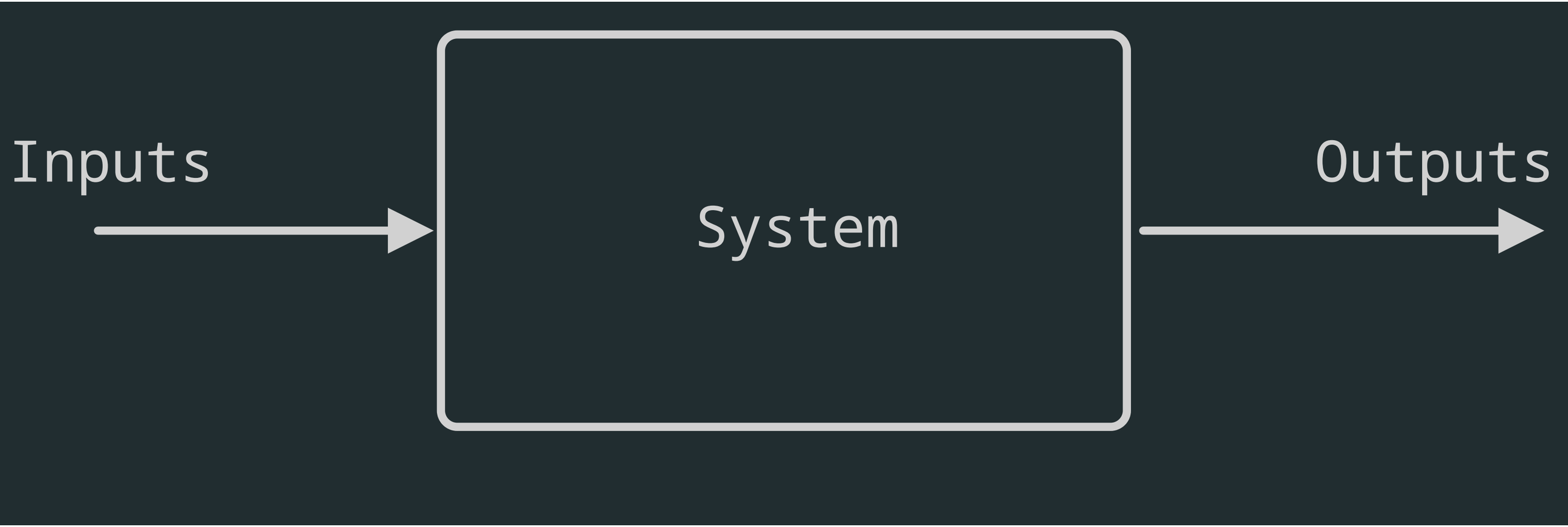
source : thinkbeforecoding.com

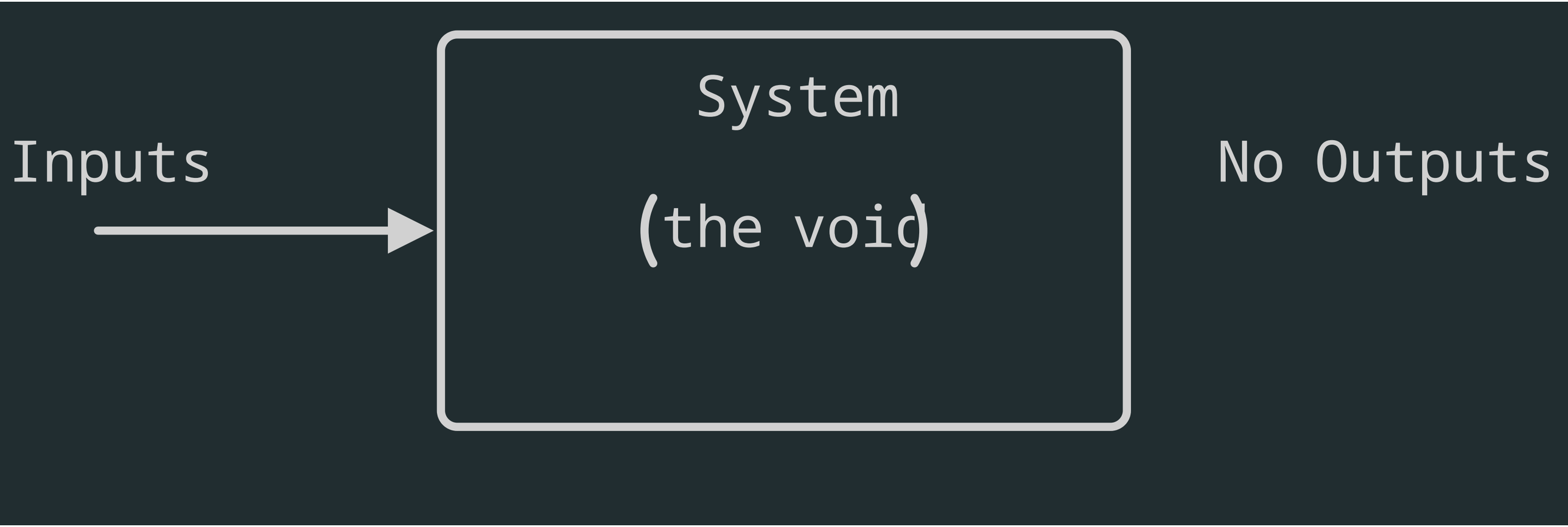

65/X100X 4X4a/X44 +1000/10
FLUX
COMPRESSION
X+YX5²/R13141 =
1614

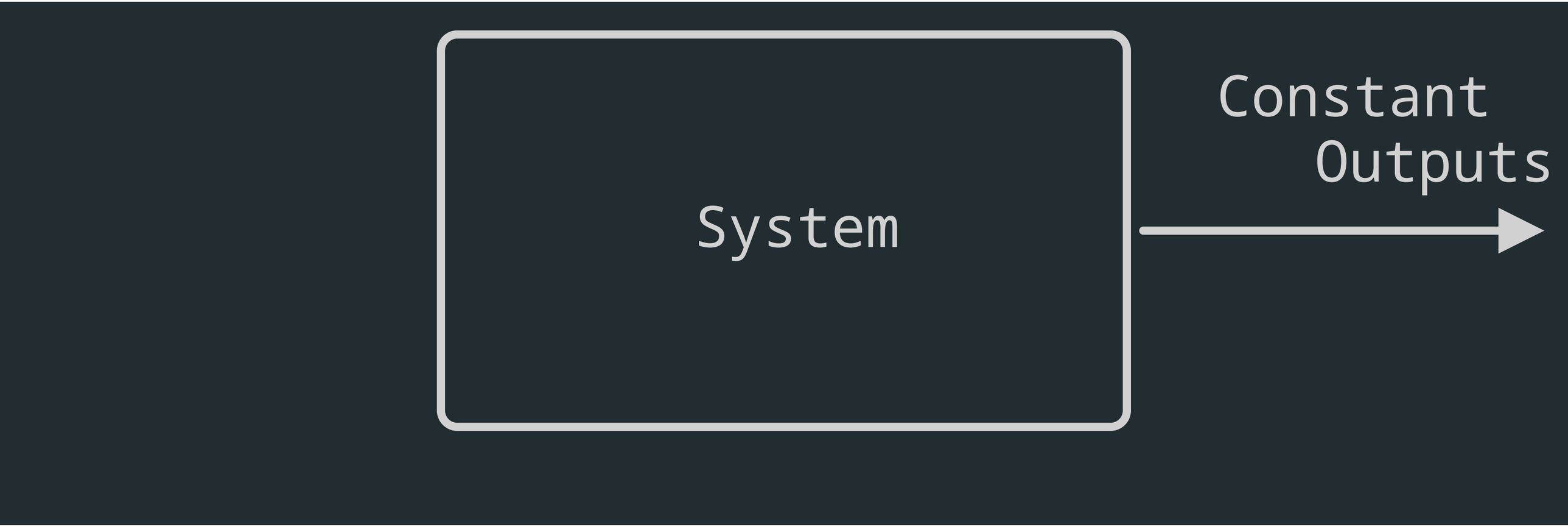

FLUI
COMPRE

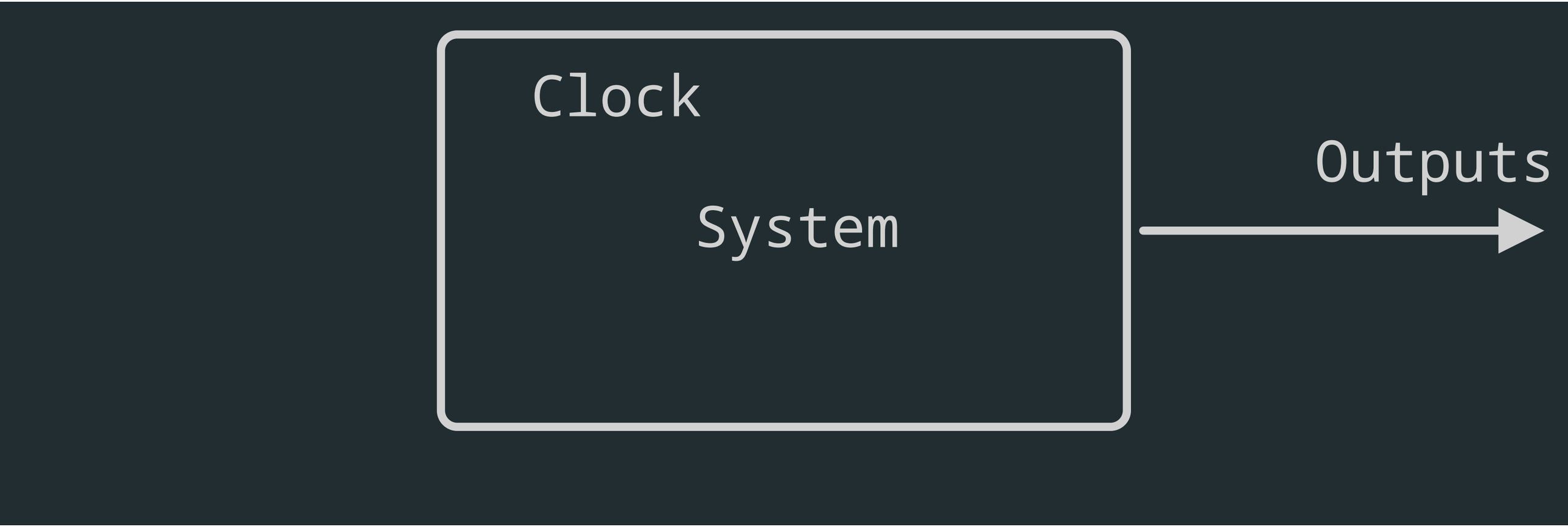
Goal for today

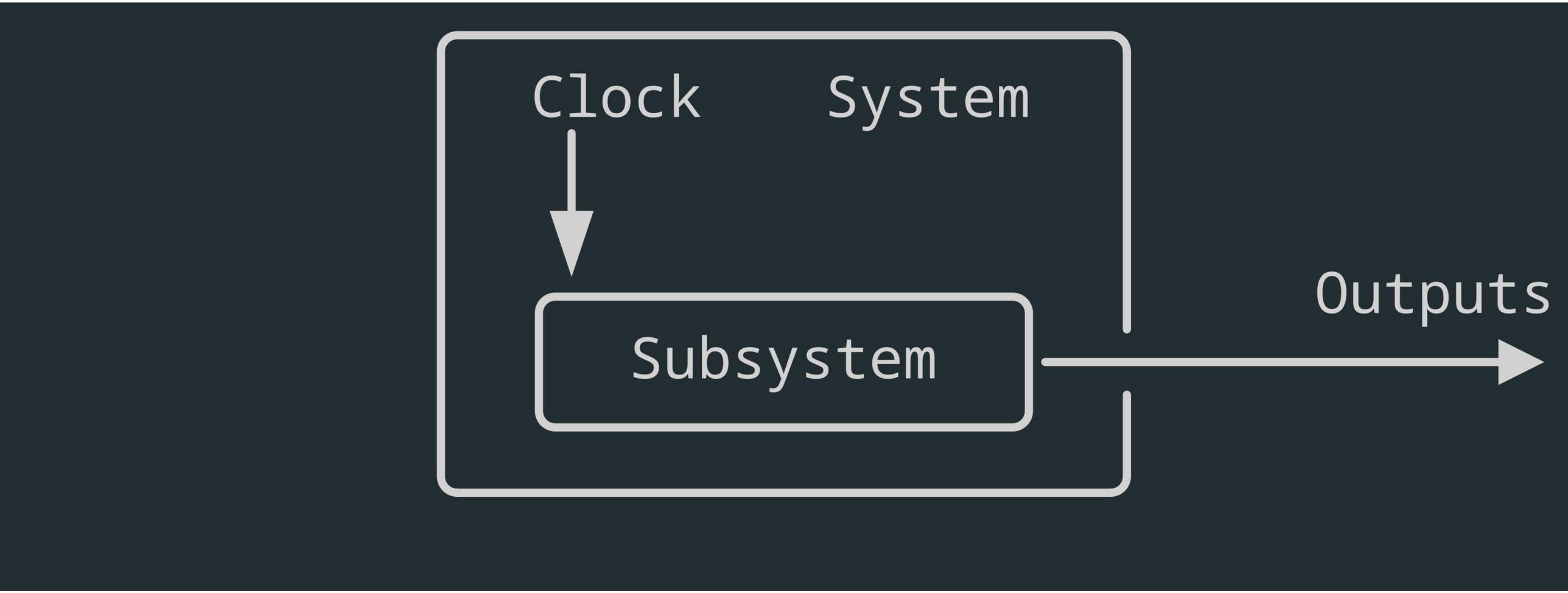
Functional Core, Imperative Shell

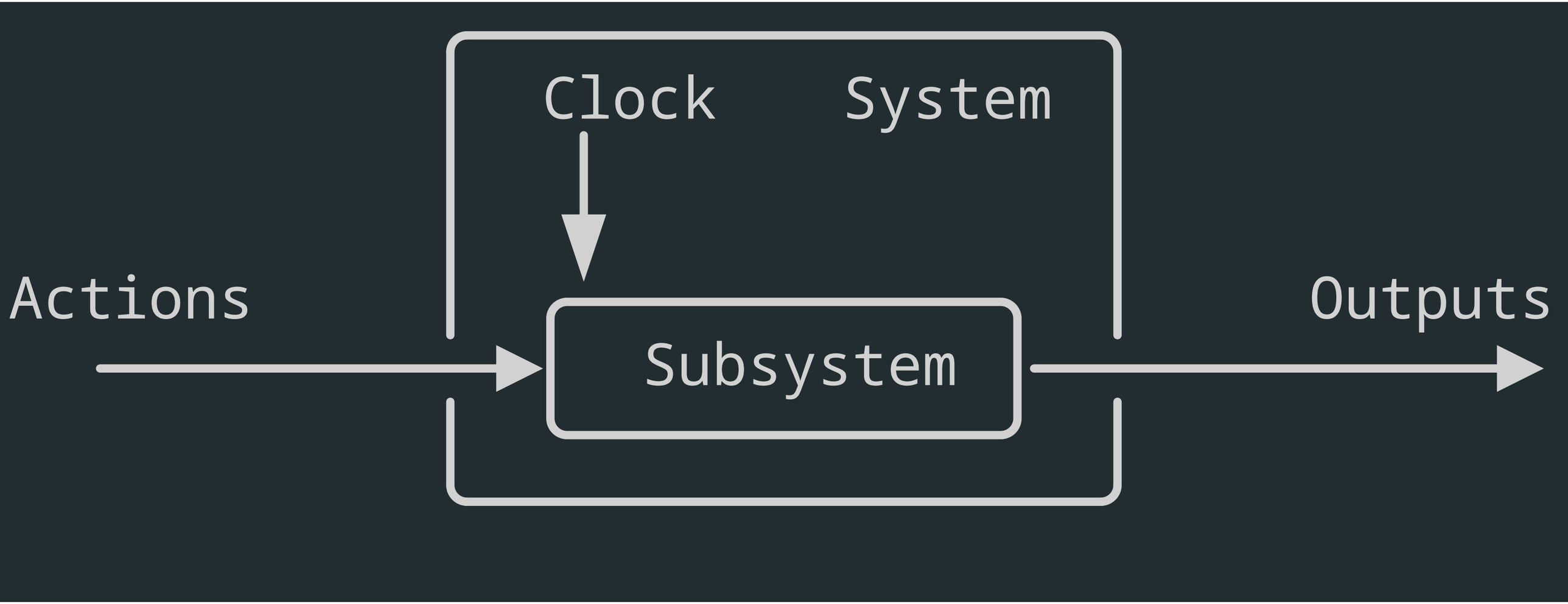

Inputs
System
Outputs

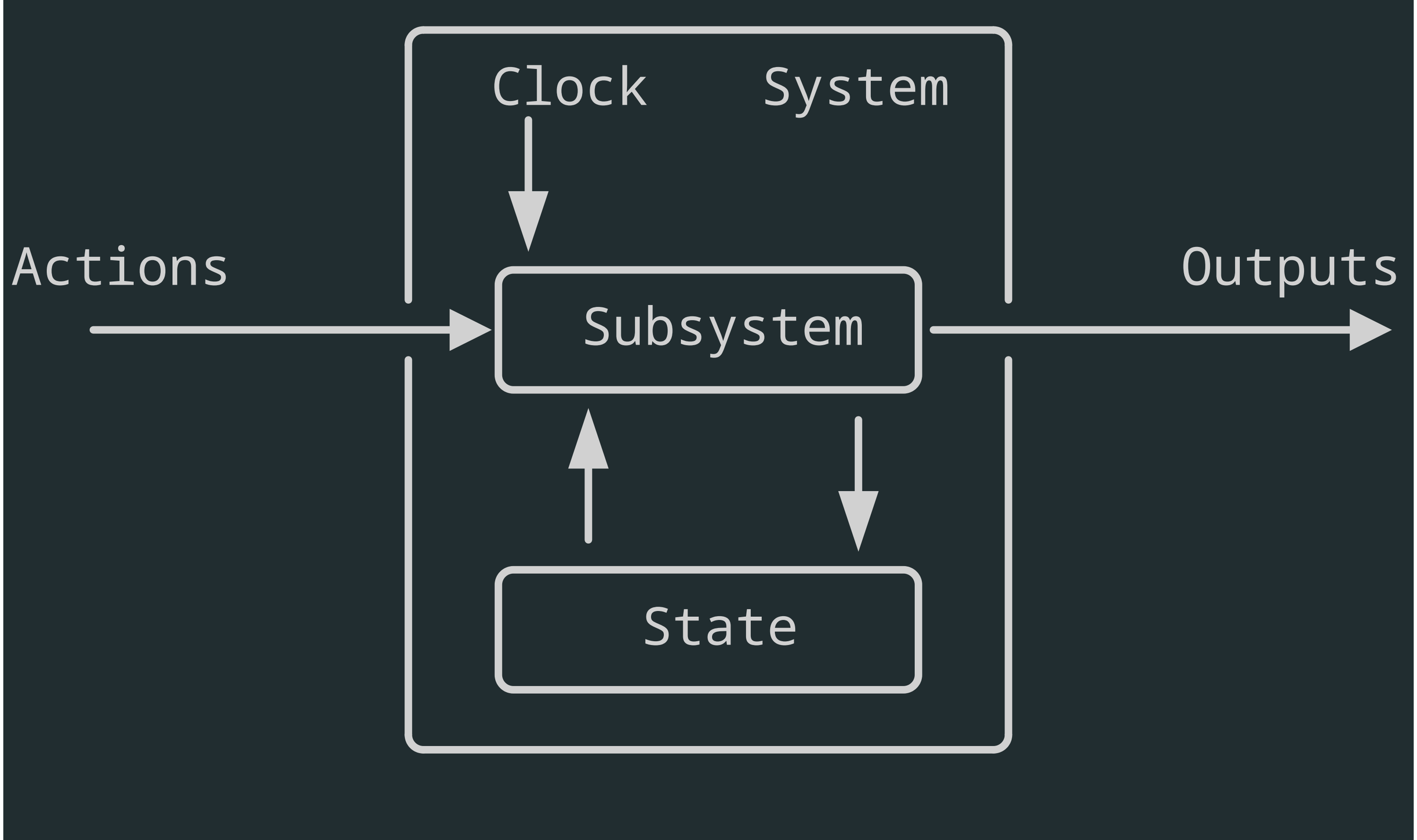

Inputs
System
(the void)
No Outputs


System
Constant
Outputs


Clock
System
Outputs

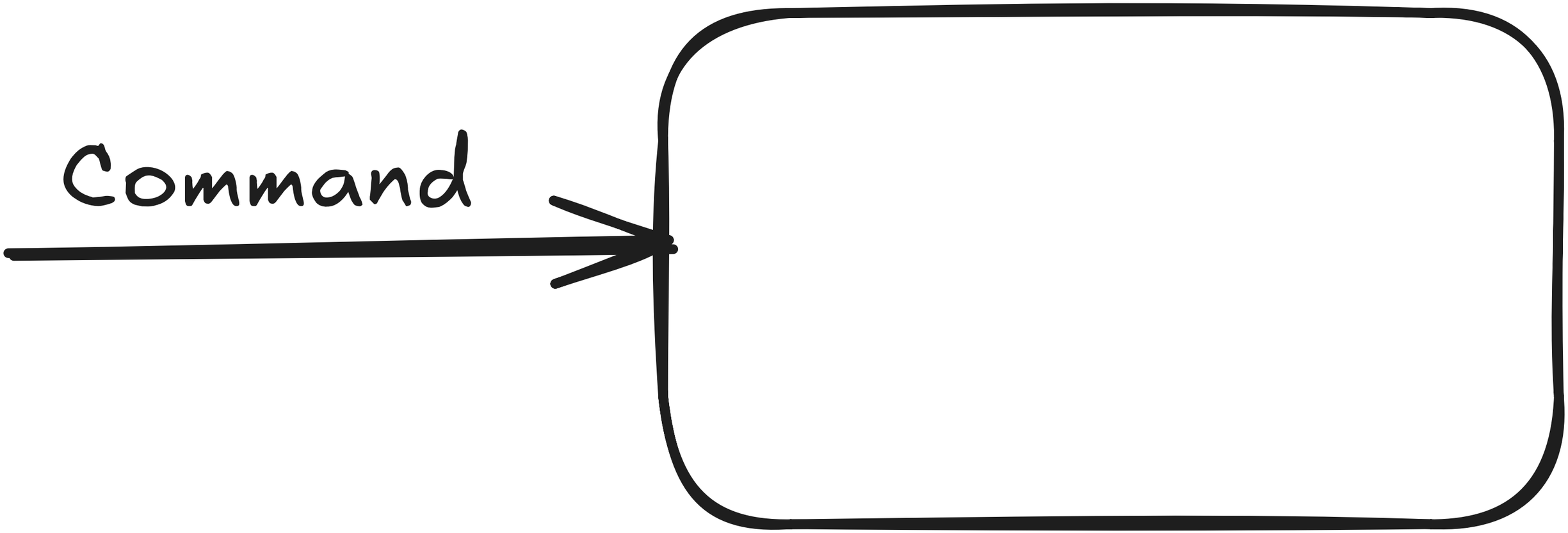

Clock
System
Subsystem
Outputs

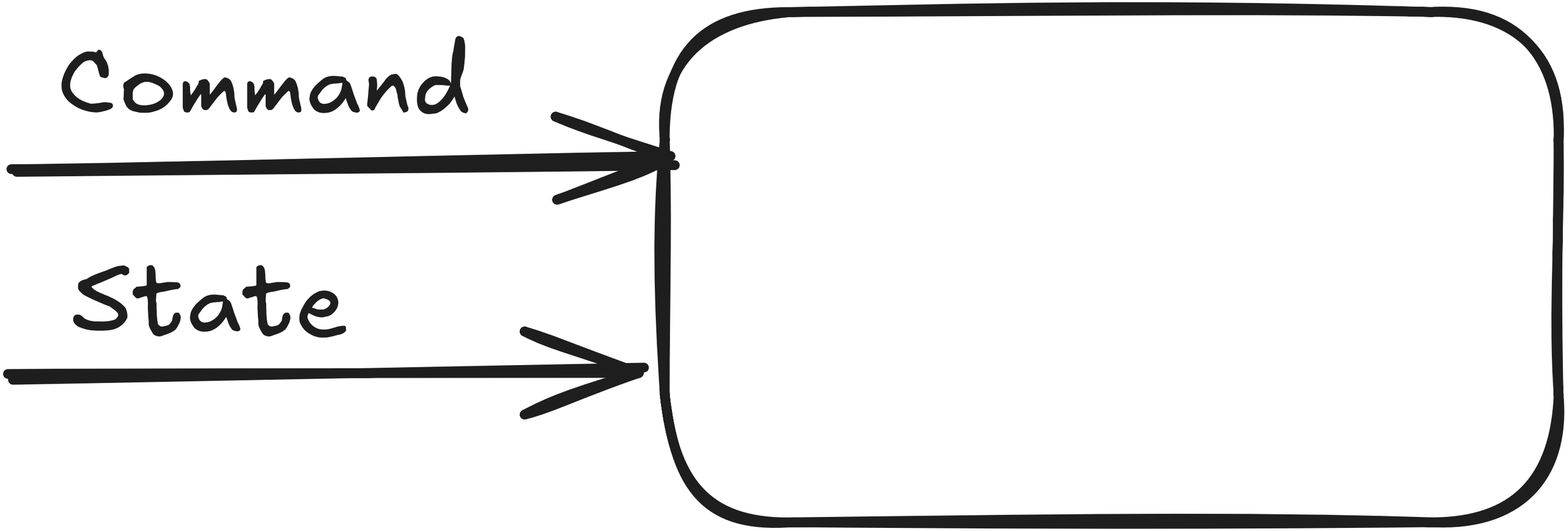

Clock
System
Actions
Subsystem
Outputs

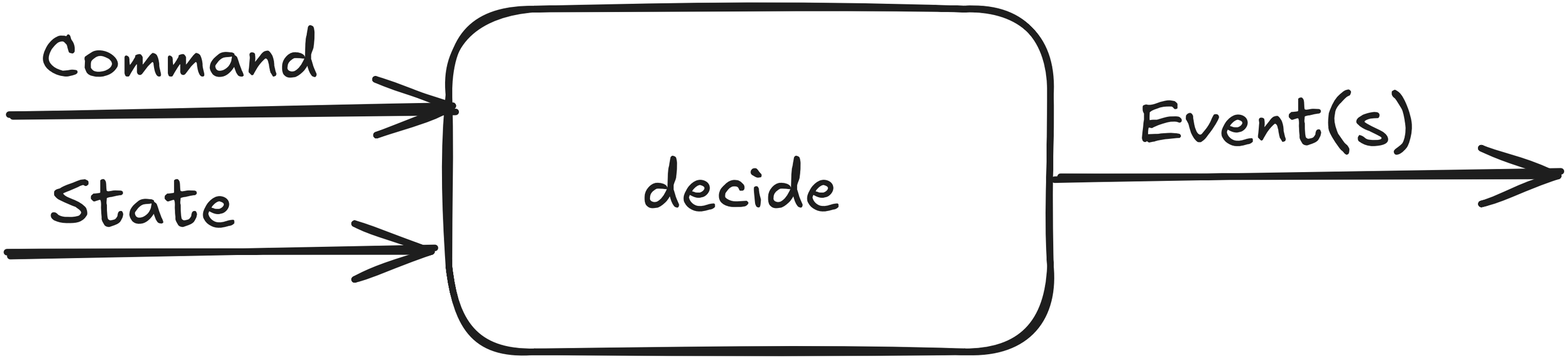

Clock
System
Actions
Outputs
Subsystem
State

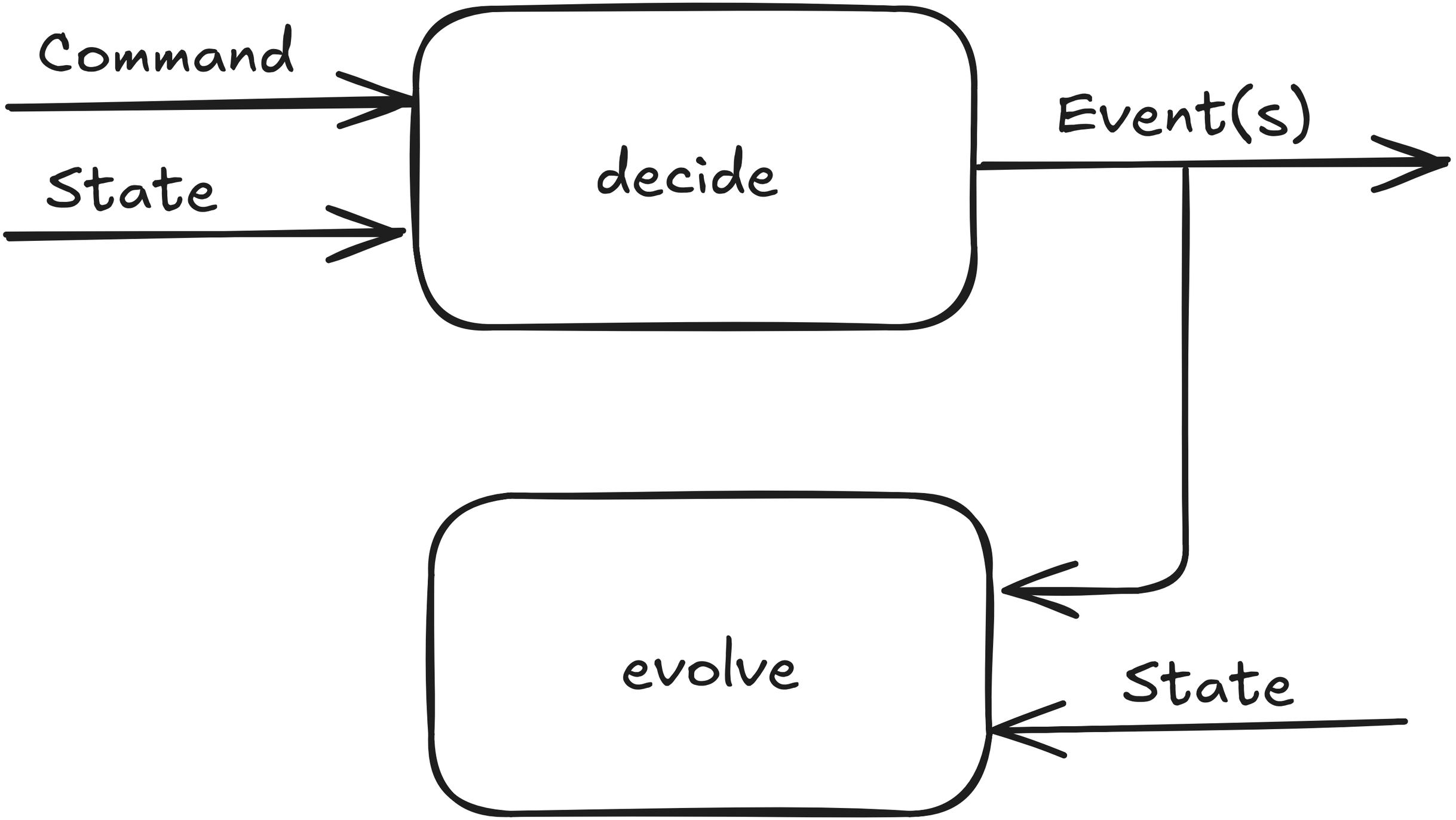
Decider

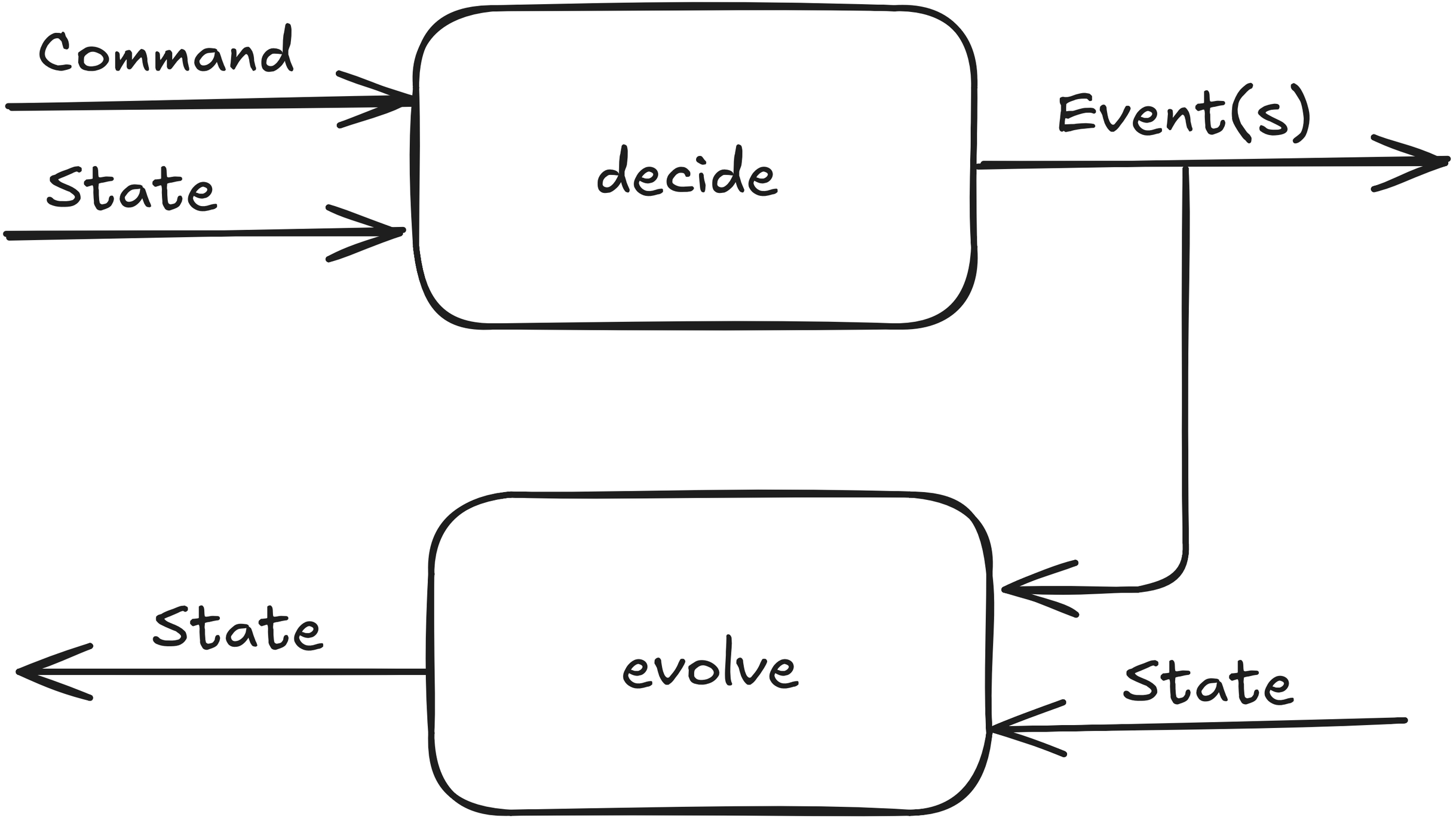
Functional Core

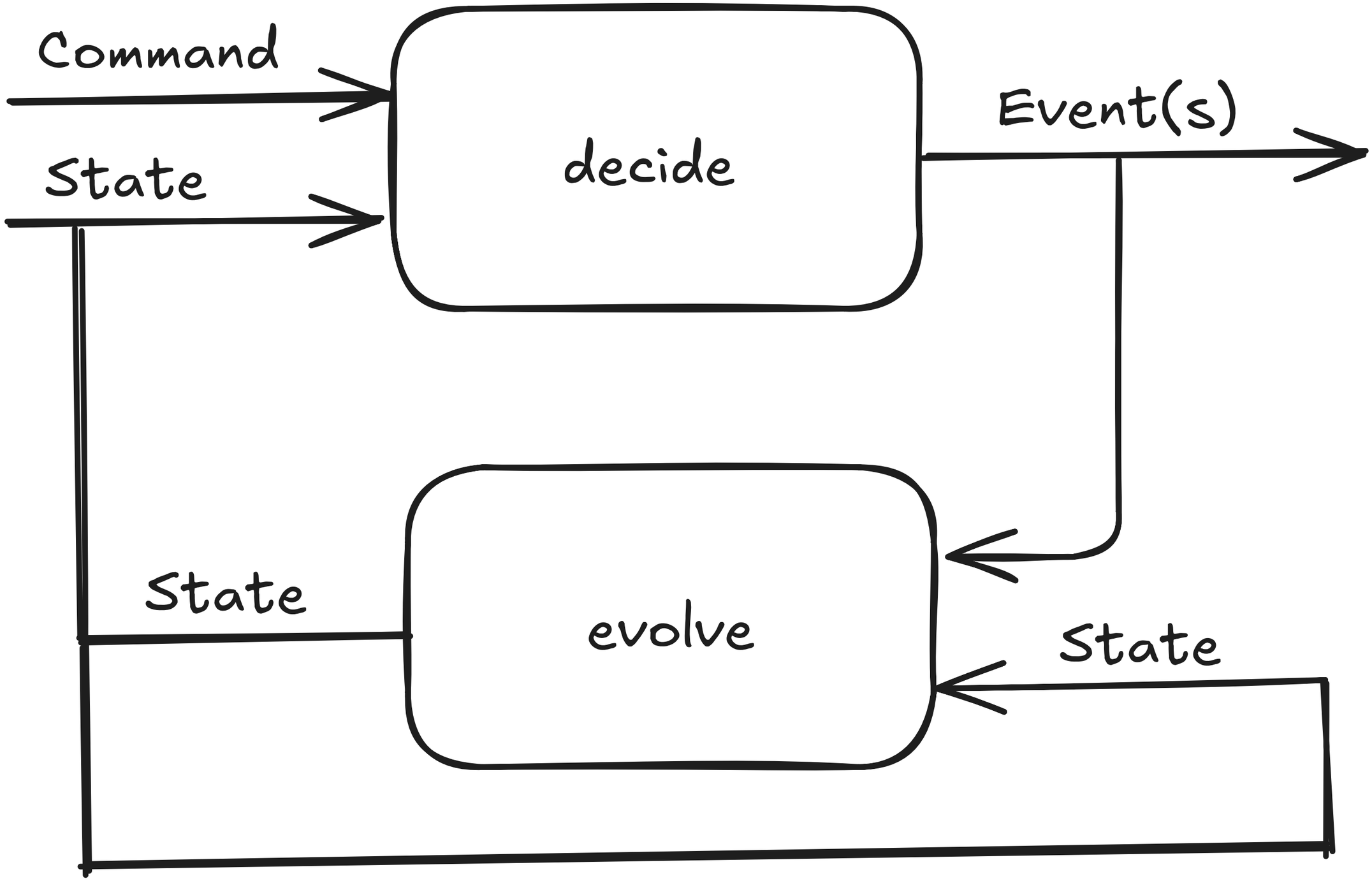

Command

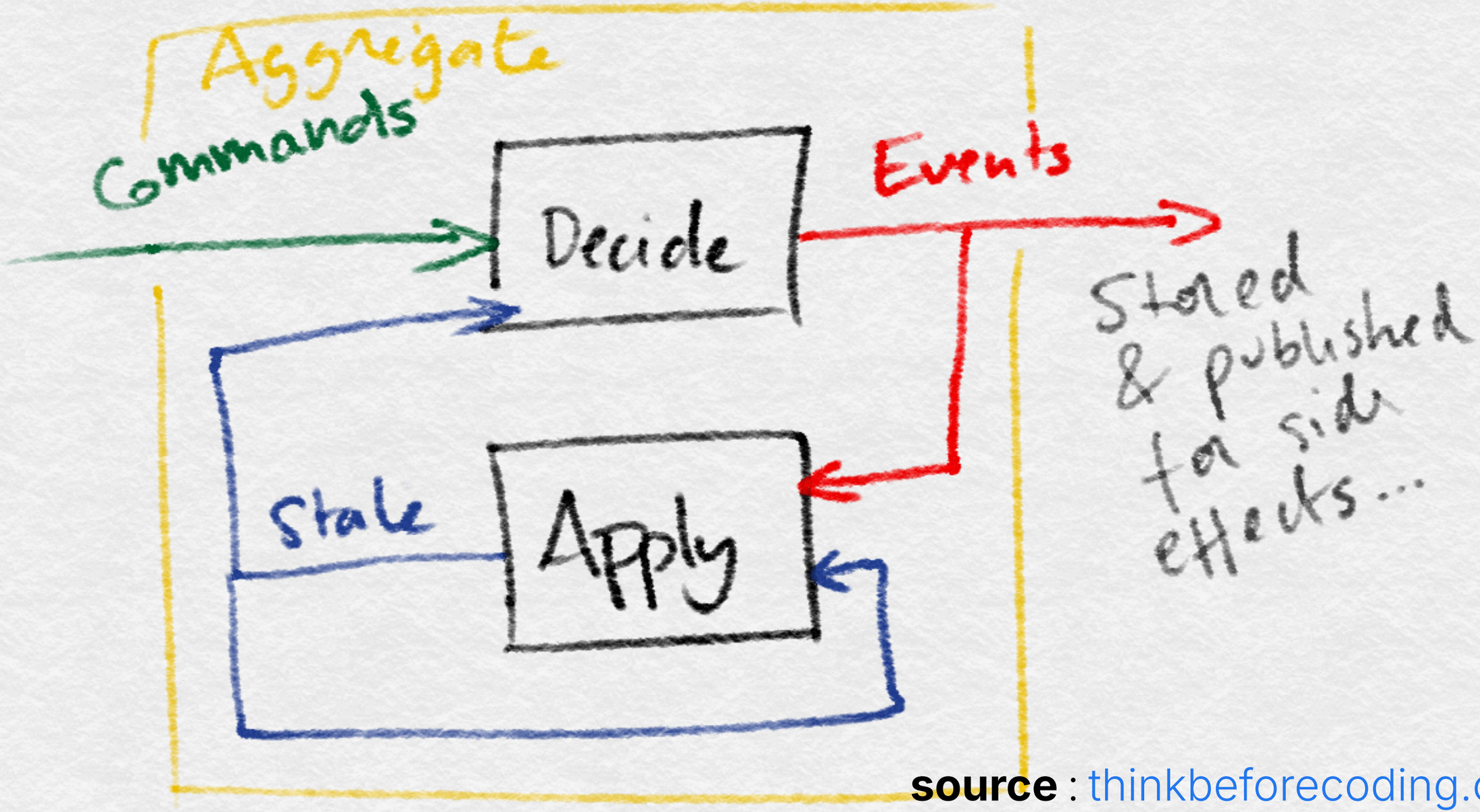

Command
State


Command
State
decide
Event(s)


Command
State
decide
Event(s)
evolve
State


Command
State
decide
Event(s)
State
evolve
State


Command
State
decide
Event(s)
State
evolve
State



source : thinkbeforecoding.com

Decider pure functions

```
def decide: [C, S, E] (C, S) → Array[E]
```

Decider pure functions

```
1  def decide: [C, S, E] (C, S) → Array[E]
2  def evolve: [S, E] (S, E) → S
```

Decider pure functions

```
1  def decide: [C, S, E] (C, S) → Array[E]
2  def evolve: [S, E] (S, E) → S
3  def initial_state: [S] () → S
```

Decider pure functions

```
def decide: [C, S, E] (C, S) → Array[E]
def evolve: [S, E] (S, E) → S
def initial_state: [S] () → S
def terminal?: [S] (S) → bool
```

Decider pure functions

```
def decide: [C, S, E] (C, S) → Array[E]
def evolve: [S, E] (S, E) → S
def initial_state: [S] () → S
def terminal?: [S] (S) → bool

# C - Command, S - State, E - Event
```

```ruby
module Toggle
  extend self

  def initial_state
  end

  def terminal?(state)
  end

  def decide(command, state)
  end

  def evolve(state, event)
  end
end
```

```
module Toggle
  extend self

  def initial_state = :off

  def terminal?(state)
  end

  def decide(command, state)
  end

  def evolve(state, event)
  end
end
```

```ruby
module Toggle
  extend self

  def initial_state = :off

  def terminal?(_state) = false

  def decide(command, state)
  end

  def evolve(state, event)
  end
end
```

```ruby
module Toggle
  extend self

  def initial_state = :off

  def terminal?(_state) = false

  def decide(command, state)
    case [command, state]
    in [:turn_on, :off]
      [:turned_on]
    end
  end

  def evolve(state, event)
  end
end
```

```ruby
module Toggle
  extend self

  def initial_state = :off

  def terminal?(_state) = false

  def decide(command, state)
    case [command, state]
    in [:turn_on, :off]
      [:turned_on]
    in [:turn_off, :on]
      [:turned_off]
    end
  end

  def evolve(state, event)
  end
end
```

```ruby
module Toggle
  extend self

  def initial_state = :off

  def terminal?(_state) = false

  def decide(command, state)
    case [command, state]
    in [:turn_on, :off]
      [:turned_on]
    in [:turn_off, :on]
      [:turned_off]
    else
      []
    end
  end

  def evolve(state, event)
  end
end
```

```ruby
module Toggle
  extend self

  def initial_state = :off

  def terminal?(_state) = false

  def decide(command, state)
    case [command, state]
    in [:turn_on, :off]
      [:turned_on]
    in [:turn_off, :on]
      [:turned_off]
    else
      []
    end
  end

  def evolve(state, event)
    case [state, event]
    in [:off, :turned_on]
      :on
    end
  end
end
```

```ruby
module Toggle
  extend self

  def decide(command, state)
    case [command, state]
    in [:turn_on, :off]
      [:turned_on]
    in [:turn_off, :on]
      [:turned_off]
    else
      []
    end
  end

  def evolve(state, event)
    case [state, event]
    in [:off, :turned_on]
      :on
    in [:on, :turned_off]
      :off
    end
  end
end
```

```ruby
module Toggle
  extend self

  def decide(command, state)
    case [command, state]
    in [:turn_on, :off]
      [:turned_on]
    in [:turn_off, :on]
      [:turned_off]
    else
      []
    end
  end

  def evolve(state, event)
    case [state, event]
    in [:off, :turned_on]
      :on
    in [:on, :turned_off]
      :off
    else
      state
    end
  end
end
```

```
state = Toggle.initial_state
# ⇒ :off

Toggle.decide(:turn_on, state)
# ⇒ [:turned_on]

Toggle.decide(:turn_off, state)
# ⇒ []

Toggle.decide(:turn_off, :on)
# ⇒ [:turned_off]
```

```ruby
state = Toggle.initial_state
# ⇒ :off

Toggle.decide(:turn_on, state)
# ⇒ [:turned_on]

Toggle.decide(:turn_off, state)
# ⇒ []

Toggle.decide(:turn_off, :on)
# ⇒ [:turned_off]
```

```ruby
state = Toggle.initial_state
# ⇒ :off

Toggle.decide(:turn_on, state)
# ⇒ [:turned_on]

Toggle.decide(:turn_off, state)
# ⇒ []

Toggle.decide(:turn_off, :on)
# ⇒ [:turned_off]
```

```
state = Toggle.initial_state
# ⇒ :off

Toggle.decide(:turn_on, state)
# ⇒ [:turned_on]

Toggle.decide(:turn_off, state)
# ⇒ []

Toggle.decide(:turn_off, :on)
# ⇒ [:turned_off]
```

```
state = Toggle.initial_state
# ⇒ :off

Toggle.decide(:turn_on, state)
# ⇒ [:turned_on]

Toggle.decide(:turn_off, state)
# ⇒ []

Toggle.decide(:turn_off, :on)
# ⇒ [:turned_off]
```

```
state = Toggle.initial_state
# ⇒ :off

Toggle.decide(:turn_on, state)
# ⇒ [:turned_on]

Toggle.decide(:turn_off, state)
# ⇒ []

Toggle.decide(:turn_off, :on)
# ⇒ [:turned_off]
```

```ruby
Toggle.evolve(:off, :turned_on)
# ⇒ :on

events.reduce(state) { |state, event|
  Toggle.evolve(state, event)
}

events.reduce(Toggle.initial_state) { |state, event|
  Toggle.evolve(state, event)
}

[:turned_on, :turned_off, :turned_on].reduce(:off) { |state, event|
  # :off, :turned_on
  # :on, :turned_off
  # :off, :turned_on
  Toggle.evolve(state, event)
}
# ⇒ :on

# shorter syntax
events.reduce(state, &Toggle.method(:evolve))
```

```ruby
Toggle.evolve(:off, :turned_on)
# ⇒ :on

events.reduce(state) { |state, event|
  Toggle.evolve(state, event)
}

events.reduce(Toggle.initial_state) { |state, event|
  Toggle.evolve(state, event)
}

[:turned_on, :turned_off, :turned_on].reduce(:off) { |state, event|
  # :off, :turned_on
  # :on, :turned_off
  # :off, :turned_on
  Toggle.evolve(state, event)
}
# ⇒ :on

# shorter syntax
events.reduce(state, &Toggle.method(:evolve))
```

```ruby
Toggle.evolve(:off, :turned_on)
# ⇒ :on

events.reduce(state) { |state, event|
  Toggle.evolve(state, event)
}

events.reduce(Toggle.initial_state) { |state, event|
  Toggle.evolve(state, event)
}

[:turned_on, :turned_off, :turned_on].reduce(:off) { |state, event|
  # :off, :turned_on
  # :on, :turned_off
  # :off, :turned_on
  Toggle.evolve(state, event)
}
# ⇒ :on

# shorter syntax
events.reduce(state, &Toggle.method(:evolve))
```

```ruby
Toggle.evolve(:off, :turned_on)
# ⇒ :on

events.reduce(state) { |state, event|
  Toggle.evolve(state, event)
}

events.reduce(Toggle.initial_state) { |state, event|
  Toggle.evolve(state, event)
}

[:turned_on, :turned_off, :turned_on].reduce(:off) { |state, event|
  # :off, :turned_on
  # :on, :turned_off
  # :off, :turned_on
  Toggle.evolve(state, event)
}
# ⇒ :on

# shorter syntax
events.reduce(state, &Toggle.method(:evolve))
```

```ruby
Toggle.evolve(:off, :turned_on)
# ⇒ :on

events.reduce(state) { |state, event|
  Toggle.evolve(state, event)
}

events.reduce(Toggle.initial_state) { |state, event|
  Toggle.evolve(state, event)
}

[:turned_on, :turned_off, :turned_on].reduce(:off) { |state, event|
  # :off, :turned_on
  # :on, :turned_off
  # :off, :turned_on
  Toggle.evolve(state, event)
}
# ⇒ :on

# shorter syntax
events.reduce(state, &Toggle.method(:evolve))
```

```ruby
Toggle.evolve(:off, :turned_on)
# ⇒ :on

events.reduce(state) { |state, event|
  Toggle.evolve(state, event)
}

events.reduce(Toggle.initial_state) { |state, event|
  Toggle.evolve(state, event)
}

[:turned_on, :turned_off, :turned_on].reduce(:off) { |state, event|
  # :off, :turned_on
  # :on, :turned_off
  # :off, :turned_on
  Toggle.evolve(state, event)
}
# ⇒ :on

# shorter syntax
events.reduce(state, &Toggle.method(:evolve))
```

```ruby
Toggle.evolve(:off, :turned_on)
# ⇒ :on

events.reduce(state) { |state, event|
  Toggle.evolve(state, event)
}

events.reduce(Toggle.initial_state) { |state, event|
  Toggle.evolve(state, event)
}

[:turned_on, :turned_off, :turned_on].reduce(:off) { |state, event|
  # :off, :turned_on
  # :on, :turned_off
  # :off, :turned_on
  Toggle.evolve(state, event)
}
# ⇒ :on

# shorter syntax
events.reduce(state, &Toggle.method(:evolve))
```

decide: [C, S, E] (C, S) → Array[E]

```ruby
# Success
def decide(command, state)
  # 0
  []
  # 1
  [SuccessEvent]
  # or many
  [SuccessEvent, SuccessEvent]
end
```

decide: [C, S, E] (C, S) → Array[E]

```
# Failure
def decide(command, state)
  []
  # or
  raise
  # or
  [FailureEvent]
end
```

evolve: [S, E] (S, E) → S

```ruby
def evolve(state, event)
  State.new(foo: event.foo)
  # or
  state.with(foo: event.foo)
  # or
  :foo
end
```

evolve: [S, E] (S, E) → S

```ruby
# Enumerable#reduce(initial_operand) { |memo, operand| ... }

[events].reduce(Decider.initial_state) { |state, event| Decider.evolve(state, event) }
# or
[events].reduce(loaded_state) { |state, event| Decider.evolve(state, event) }
# or
[events].reduce(state, &Decider.method(:evolve))
```

Decider is a Monoid

In abstract algebra (...), a monoid is a set equipped with an associative binary operation and an identity element.

Wikipedia

$$*\colon S \times S \to S$$

$$\forall_{a \in S}\ e * a = a * e = a$$

$$\forall_{a,b,c \in S}\ (a * b) * c = a * (b * c)$$

```
def compose: (D[Ca, Sa, Ea], D[Cb, Sb, Eb]) → D[Ca | Cb, Sa * Sb, Ea | Eb]
```

```
def compose: (D[Ca, Sa, Ea], D[Cb, Sb, Eb]) → D[Ca | Cb, Sa * Sb, Ea | Eb]

def initial_state: () → [Sa, Sb]
```

```
def compose: (D[Ca, Sa, Ea], D[Cb, Sb, Eb]) → D[Ca | Cb, Sa * Sb, Ea | Eb]
def initial_state: () → [Sa, Sb]

def terminal?: ([Sa, Sb]) → terminal?(Sa) && terminal?(Sb)
```

```
def compose: (D[Ca, Sa, Ea], D[Cb, Sb, Eb]) → D[Ca | Cb, Sa * Sb, Ea | Eb]
def initial_state: () → [Sa, Sb]
def terminal?: ([Sa, Sb]) → terminal?(Sa) && terminal?(Sb)

def decide: (Ca | Cb, [Sa, Sb]) → [Ea | Eb]
```

```
def compose: (D[Ca, Sa, Ea], D[Cb, Sb, Eb]) → D[Ca | Cb, Sa * Sb, Ea | Eb]
def initial_state: () → [Sa, Sb]
def terminal?: ([Sa, Sb]) → terminal?(Sa) && terminal?(Sb)
def decide: (Ca | Cb, [Sa, Sb]) → [Ea | Eb]

def evolve: ([Sa, Sb], Ea | Eb) → [Sa', Sb] | [Sa, Sb']
```

gem install decider

15 commits, version 2.0, 2009

```ruby
module Decider
  extend self

  def decide(*choices)
    choices.shuffle[0]
  end
end
```

gem install decide.rb

```ruby
Toggle = Decider.define do
  initial_state :off

  terminal? { state == :blown }

  decide :turn_on, :off do
    emit :turned_on
  end

  decide :turn_off, :on do
    emit :turned_on
  end

  evolve :turned_on do
    :on
  end

  evolve :turned_off do
    :off
  end
end

Composition = Decider.compose(Foo, Bar)
```

```ruby
Toggle = Decider.define do
  initial_state :off

  terminal? { state == :blown }

  decide :turn_on, :off do
    emit :turned_on
  end

  decide :turn_off, :on do
    emit :turned_on
  end

  evolve :turned_on do
    :on
  end

  evolve :turned_off do
    :off
  end
end

Composition = Decider.compose(Foo, Bar)
```

```
Toggle = Decider.define do
  initial_state :off

  terminal? { state == :blown }

  decide :turn_on, :off do
    emit :turned_on
  end

  decide :turn_off, :on do
    emit :turned_on
  end

  evolve :turned_on do
    :on
  end

  evolve :turned_off do
    :off
  end
end

Composition = Decider.compose(Foo, Bar)
```

```
Toggle = Decider.define do
  initial_state :off

  terminal? { state == :blown }

  decide :turn_on, :off do
    emit :turned_on
  end

  decide :turn_off, :on do
    emit :turned_on
  end

  evolve :turned_on do
    :on
  end

  evolve :turned_off do
    :off
  end
end

Composition = Decider.compose(Foo, Bar)
```

```
Toggle = Decider.define do
  initial_state :off

  terminal? { state == :blown }

  decide :turn_on, :off do
    emit :turned_on
  end

  decide :turn_off, :on do
    emit :turned_on
  end

  evolve :turned_on do
    :on
  end

  evolve :turned_off do
    :off
  end
end

Composition = Decider.compose(Foo, Bar)
```

```
Toggle = Decider.define do
  initial_state :off

  terminal? { state == :blown }

  decide :turn_on, :off do
    emit :turned_on
  end

  decide :turn_off, :on do
    emit :turned_on
  end

  evolve :turned_on do
    :on
  end

  evolve :turned_off do
    :off
  end
end

Composition = Decider.compose(Foo, Bar)
```

```ruby
Toggle = Decider.define do
  initial_state :off

  terminal? { state == :blown }

  decide :turn_on, :off do
    emit :turned_on
  end

  decide :turn_off, :on do
    emit :turned_on
  end

  evolve :turned_on do
    :on
  end

  evolve :turned_off do
    :off
  end
end

Composition = Decider.compose(Foo, Bar)
```

Testing

```
class TestBulbDecider < Minitest::Test
  def decider = BulbDecider

  def given(events)
    @state = events.reduce(
      decider.initial_state, &decider.method(:evolve)
    )

    self
  end

  def when(command)
    @events = decider.decide(command, @state)

    self
  end

  def expect(expected) = assert_equal(expected, @events)
end
```

Testing

```
class TestBulbDecider < Minitest::Test
  def decider = BulbDecider

  def given(events)
    @state = events.reduce(
      decider.initial_state, &decider.method(:evolve)
    )

    self
  end

  def when(command)
    @events = decider.decide(command, @state)

    self
  end

  def expect(expected) = assert_equal(expected, @events)
end
```

Testing

```ruby
class TestBulbDecider < Minitest::Test
  def decider = BulbDecider

  def given(events)
    @state = events.reduce(
      decider.initial_state, &decider.method(:evolve)
    )

    self
  end

  def when(command)
    @events = decider.decide(command, @state)

    self
  end

  def expect(expected) = assert_equal(expected, @events)
end
```

Testing

```
class TestBulbDecider < Minitest::Test
  def decider = BulbDecider

  def given(events)
    @state = events.reduce(
      decider.initial_state, &decider.method(:evolve)
    )

    self
  end

  def when(command)
    @events = decider.decide(command, @state)

    self
  end

  def expect(expected) = assert_equal(expected, @events)
end
```

Testing

```
class TestBulbDecider < Minitest::Test
  def decider = BulbDecider

  def given(events)
    @state = events.reduce(
      decider.initial_state, &decider.method(:evolve)
    )

    self
  end

  def when(command)
    @events = decider.decide(command, @state)

    self
  end

  def expect(expected) = assert_equal(expected, @events)
end
```

Testing

```
class TestBulbDecider < Minitest::Test
  def decider = BulbDecider

  def given(events)
    @state = events.reduce(
      decider.initial_state, &decider.method(:evolve)
    )

    self
  end

  def when(command)
    @events = decider.decide(command, @state)

    self
  end

  def expect(expected) = assert_equal(expected, @events)
end
```

Testing

```
class TestBulbDecider < Minitest::Test
  def test_fit
    given(
      []
    ).when(
      FitBulb.new(max_uses: 5)
    ).expect(
      [BulbWasFitted.new(max_uses: 5)]
    )
  end
end
```

Testing

```
class TestBulbDecider < Minitest::Test
  def test_switch_on
    given(
      [BulbWasFitted.new(max_uses: 5)]
    ).when(
      SwitchBulbOn.new
    ).expect(
      [BulbWasSwitchedOn.new]
    )
  end
end
```

Testing

```
class TestBulbDecider < Minitest::Test
  def test_blow
    given(
      [
        BulbWasFitted.new(max_uses: 2),
        BulbWasSwitchedOn.new,
        BulbWasSwitchedOff.new,
        BulbWasSwitchedOn.new,
        BulbWasSwitchedOff.new
      ]
    ).when(
      SwitchBulbOn.new
    ).expect(
      [BulbWasBlown.new]
    )
  end
end
```

Reframing the event

An event represents a meaningful business decission taken, which alters the state of the system.

Yves Goevelen

Infrastructure

In Memory

```
class CommandHandler
  def initialize
    @states = Hash.new { |h, k| h[k] = CommandDecider.initial_state }
  end

  def call(id, command)
    events = CommandDecider.decide(command, @states[id])

    @states[id] = events.reduce(@states[id], &CommandDecider.evolve)

    events
  end
end
```

In Memory

```ruby
class CommandHandler
  def initialize
    @states = Hash.new { |h, k| h[k] = CommandDecider.initial_state }
  end

  def call(id, command)
    events = CommandDecider.decide(command, @states[id])

    @states[id] = events.reduce(@states[id], &CommandDecider.evolve)

    events
  end
end
```

In Memory

```
class CommandHandler
  def initialize
    @states = Hash.new { |h, k| h[k] = CommandDecider.initial_state }
  end

  def call(id, command)
    events = CommandDecider.decide(command, @states[id])

    @states[id] = events.reduce(@states[id], &CommandDecider.evolve)

    events
  end
end
```

State

```
class CommandHandler
  def initialize
    @repository = Repository.new
  end

  def call(id, command)
    state = @repository.load(id)

    events = CommandDecider.decide(command, state)

    @repository.save(id, events.reduce(state, &CommandDecider.evolve))

    events
  end
end
```

State

```
class CommandHandler
  def initialize
    @repository = Repository.new
  end

  def call(id, command)
    state = @repository.load(id)

    events = CommandDecider.decide(command, state)

    @repository.save(id, events.reduce(state, &CommandDecider.evolve))

    events
  end
end
```

State

```
class CommandHandler
  def initialize
    @repository = Repository.new
  end

  def call(id, command)
    state = @repository.load(id)

    events = CommandDecider.decide(command, state)

    @repository.save(id, events.reduce(state, &CommandDecider.evolve))

    events
  end
end
```

Event Sourcing

```
class CommandHandler
  def initialize
    @event_store = EventStore.new
  end

  def call(id, command)
    events = @event_store.load_stream(id)
    state = events.reduce(
      CommandDecider.initial_state, &CommandDecider.evolve
    )

    events = CommandDecider.decide(command, state)
    @event_store.append_to_stream(id, events)

    events
  end
end
```

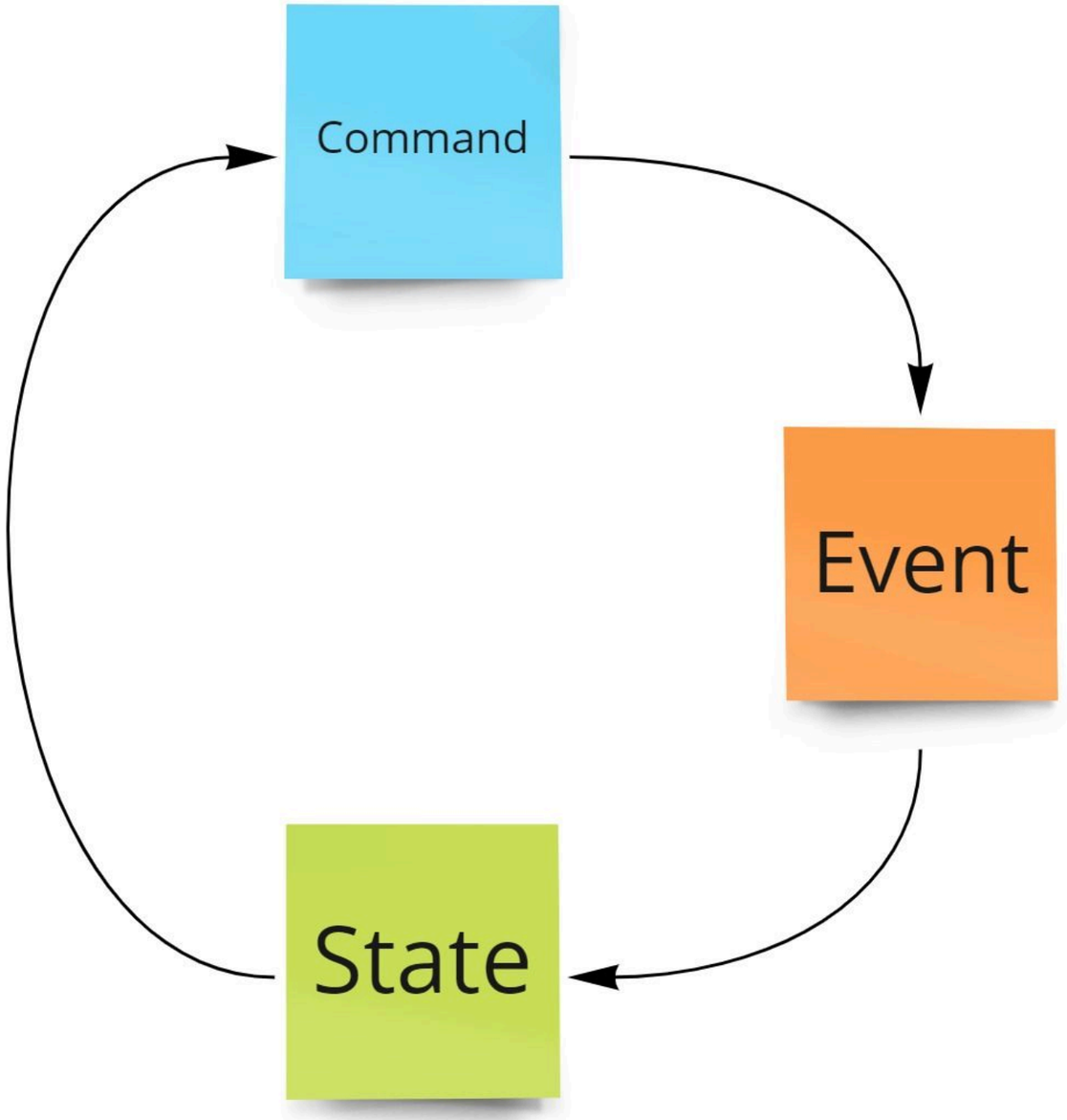
Event Sourcing

```ruby
class CommandHandler
  def initialize
    @event_store = EventStore.new
  end

  def call(id, command)
    events = @event_store.load_stream(id)
    state = events.reduce(
      CommandDecider.initial_state, &CommandDecider.evolve
    )

    events = CommandDecider.decide(command, state)
    @event_store.append_to_stream(id, events)

    events
  end
end
```

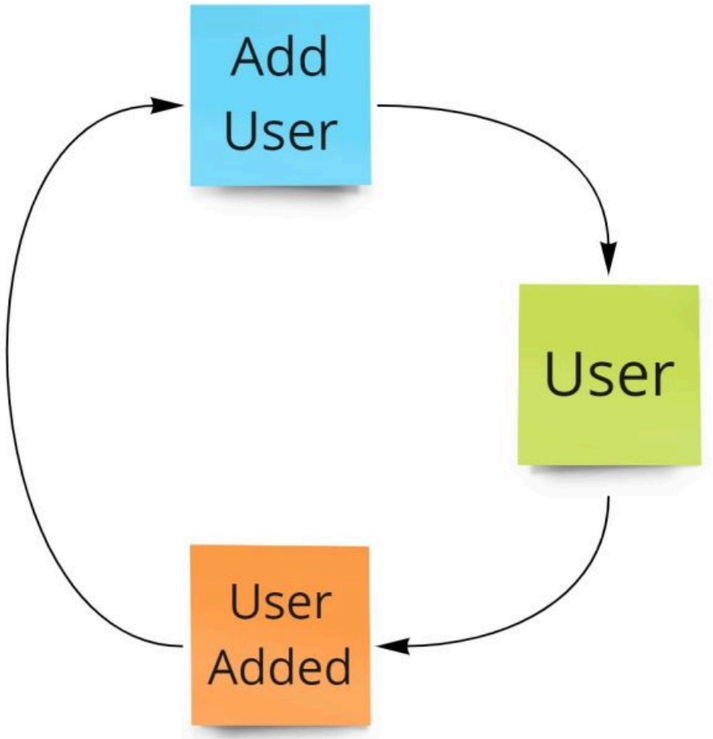
Event Sourcing

```
class CommandHandler
  def initialize
    @event_store = EventStore.new
  end

  def call(id, command)
    events = @event_store.load_stream(id)
    state = events.reduce(
      CommandDecider.initial_state, &CommandDecider.evolve
    )

    events = CommandDecider.decide(command, state)
    @event_store.append_to_stream(id, events)

    events
  end
end
```

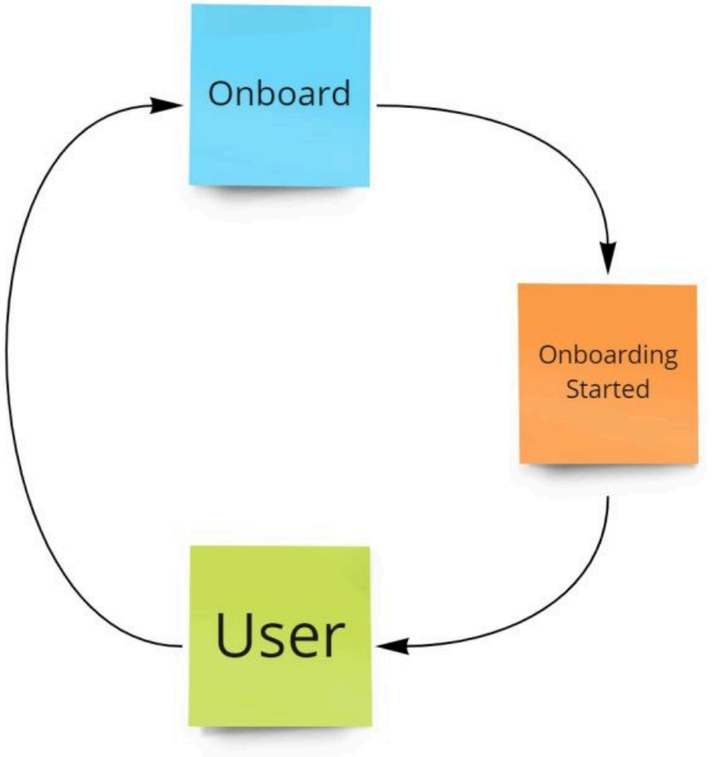
Infrastructure does not impact domain

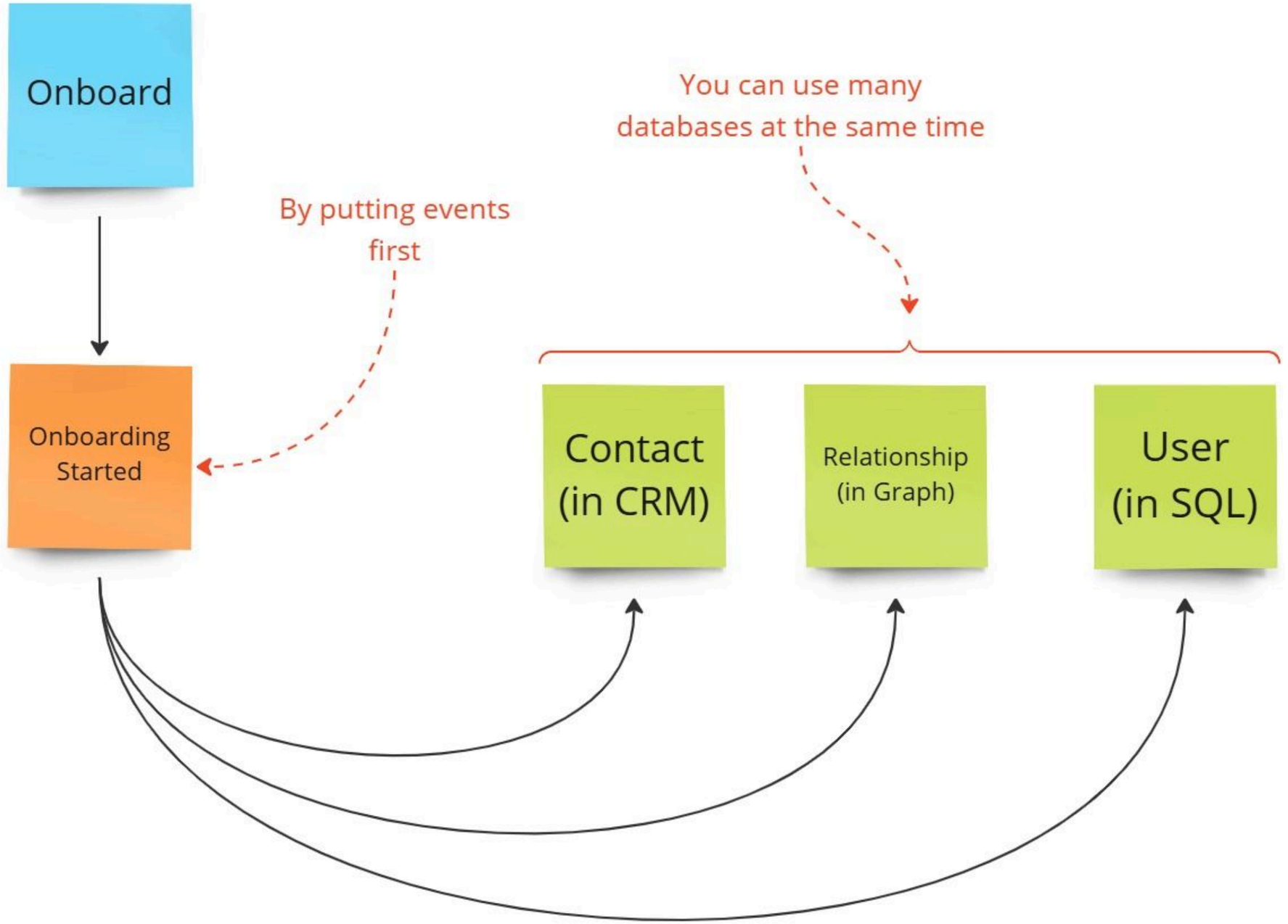

Command
Event
State

✗ Not this



✓ This




Onboard
Onboarding Started
By putting events first
You can use many databases at the same time
Contact (in CRM)
Relationship (in Graph)
User (in SQL)

Road to perfect DSL

Pattern matching

```ruby
def decide(command, state)
  case [command, state]
  in [CreateIssue, State(id:, status: nil)]
    [IssueOpened.new(data: { issue_id: id })]
  in [ResolveIssue, State(id:, status: :open | :in_progress | :reopened)]
    [IssueResolved.new(data: { issue_id: id })]
  in [CloseIssue, State(id:, status: :open | :in_progress | :resolved | :reopened)]
    [IssueClosed.new(data: { issue_id: id })]
  in [ReopenIssue, State(id:, status: :resolved | :closed)]
    [IssueReopened.new(data: { issue_id: id })]
  in [StartIssueProgress, State(id:, status: :open | :reopened)]
    [IssueProgressStarted.new(data: { issue_id: id })]
  in [StopIssueProgress, State(id:, status: :in_progress)]
    [IssueProgressStopped.new(data: { issue_id: id })]
  else
    [InvalidTransition.new]
  end
end
```

DSL

```
decide CreateIssue do |command, state|
  if !state.status
    [IssueOpened.new(data: { issue_id: state.id })]
  else
    [InvalidTransition.new]
  end
end

decide ResovleIssue do |command, state|
  if state.status.include? %i[open in_progress reopened]
    [IssueResolved.new(data: { issue_id: state.id })]
  else
    [InvalidTransition.new]
  end
end

# ...
```


WHY NOT BOTH?
memeger

DSL with Pattern matching

```ruby
decide CreateIssue, State(id:, status: nil) do
  [IssueOpened.new(data: { issue_id: id })]
end

decide ResovleIssue, State(id:, status: :open | :in_progress | :reopened) do
  [IssueResolved.new(data: { issue_id: state.id })]
end

decide CloseIssue, State(id:, status: :open | :in_progress | :resolved | :reopened)] do
  [IssueClosed.new(data: { issue_id: id })]
end

decide ReopenIssue, State(id:, status: :resolved | :closed)] do
  [IssueReopened.new(data: { issue_id: id })]
end

# ...
```

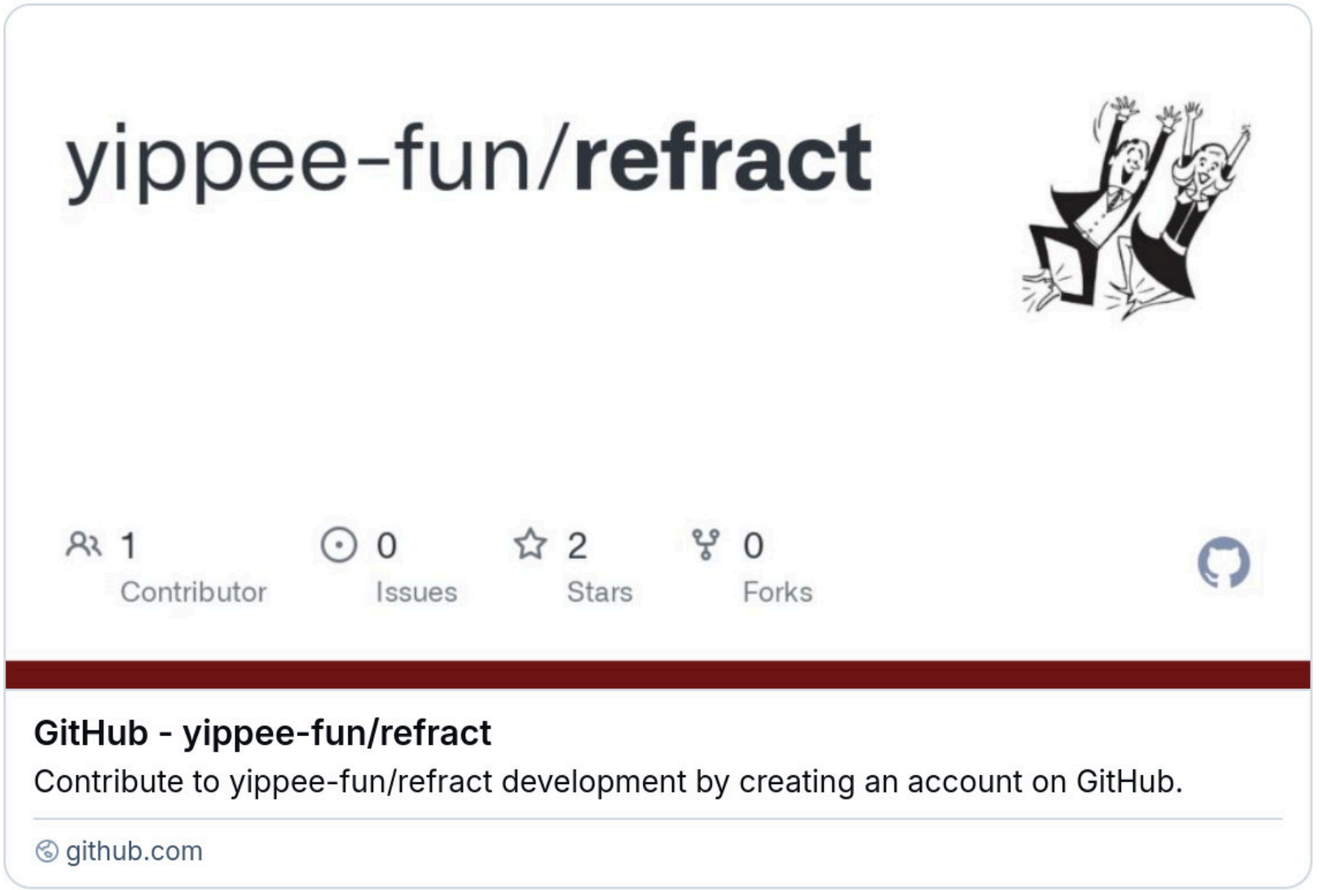
💡 Parse into `AST` and rewrite to `case` 🤔

Inspiration: https://zverok.substack.com/p/elixir-like-pipes-in-ruby-oh-no-not

```ruby
require 'not_a_pipe'

extend NotAPipe

pipe def repos(username)
  username >>
    "https://api.github.com/users/#{_}/repos" >>
    URI.open >>
    _.read >>
    JSON.parse(symbolize_names: true) >>
    _.map { _1.dig(:full_name) }.first(10) >>
    pp
end

repos('zverok')
# prints: ["zverok/any_good", "zverok/awesome-codebases", "zverok/awesome-events", "zverok/backports", ...
```

Joel Drapper 🇬🇧🇺🇦
@joel.drapper.me

Refract [WIP] is a new Ruby gem that lets you rewrite Ruby code at the AST level. It converts concrete Prism syntax trees to its own abstract syntax tree, allowing you to walk the tree, mutating and inserting nodes. Then it formats these abstract nodes back into valid Ruby code.



July 17, 2025 at 2:24 PM · Some people can reply · Translate

Extensions

Option

```
data Option a
  = Some a
  | None
```

Option

```
require "literal"

Some = Data.define(:value)
None = Data.define

Option = Literal::Types._Union(Some, None)
```

Map

```
# ((a → b), Option[a]) → Option[b]
map = →(fn, option) {
  case option
  in Some(value:)
    Some.new(value: fn.(value))
  in None
    None
  end
}
```

Map2

```
# ((a → b → c), Option[a], Option[b]) → Option[c]
map2 = →(fn, opta, optb) {
  case [opta, optb]
  in [None, _] | [_, None]
    None
  in [Some(value: a), Some(value: b)]
    Some.new(value: fn.(a, b))
  end
}
```

Go on with `map3`, `map4`, `map5`...

Does ~~Rails~~ Decider scale?

Currying

```
fn = →(a, b, c) { a + b + c }

gn = →(a) {
  →(b) {
    →(c) {
      a + b + c
    }
  }
}

fn.(1, 2, 3) == gn.(1).(2).(3)
```

Ruby has curry builtin: `fn.curry`

Apply

```
apply = →(f, x) { map2.(→(f, x) { f.(x) }, f, x) }

basic_apply = →(f, x) { f.(x) }
apply = →(f, x) { map2.(basic_apply, f, x) }

add = →(x, y, z) { x + y + z }

addx = map.(add.curry, Some.new(value: 2))
# ⇒ #<data Some value=Proc>
addy = apply.(addx, Some.new(value: 5))
# ⇒ #<data Some value=Proc>
apply.(addy, Some.new(value: 3))
# ⇒ #<data Some value=10>
```

Apply

```
apply = →(f, x) { map2.(→(f, x) { f.(x) }, f, x) }

basic_apply = →(f, x) { f.(x) }
apply = →(f, x) { map2.(basic_apply, f, x) }

add = →(x, y, z) { x + y + z }

addx = map.(add.curry, Some.new(value: 2))
# ⇒ #<data Some value=Proc>
addy = apply.(addx, Some.new(value: 5))
# ⇒ #<data Some value=Proc>
apply.(addy, Some.new(value: 3))
# ⇒ #<data Some value=10>
```

Apply

```
apply = →(f, x) { map2.(→(f, x) { f.(x) }, f, x) }

basic_apply = →(f, x) { f.(x) }
apply = →(f, x) { map2.(basic_apply, f, x) }

add = →(x, y, z) { x + y + z }

addx = map.(add.curry, Some.new(value: 2))
# ⇒ #<data Some value=Proc>
addy = apply.(addx, Some.new(value: 5))
# ⇒ #<data Some value=Proc>
apply.(addy, Some.new(value: 3))
# ⇒ #<data Some value=10>
```

Apply

```
apply = →(f, x) { map2.(→(f, x) { f.(x) }, f, x) }

basic_apply = →(f, x) { f.(x) }
apply = →(f, x) { map2.(basic_apply, f, x) }

add = →(x, y, z) { x + y + z }

addx = map.(add.curry, Some.new(value: 2))
# ⇒ #<data Some value=Proc>
addy = apply.(addx, Some.new(value: 5))
# ⇒ #<data Some value=Proc>
apply.(addy, Some.new(value: 3))
# ⇒ #<data Some value=10>
```

Apply

```
apply = →(f, x) { map2.(→(f, x) { f.(x) }, f, x) }

basic_apply = →(f, x) { f.(x) }
apply = →(f, x) { map2.(basic_apply, f, x) }

add = →(x, y, z) { x + y + z }

addx = map.(add.curry, Some.new(value: 2))
# ⇒ #<data Some value=Proc>
addy = apply.(addx, Some.new(value: 5))
# ⇒ #<data Some value=Proc>
apply.(addy, Some.new(value: 3))
# ⇒ #<data Some value=10>
```

Apply

```
apply = →(f, x) { map2.(→(f, x) { f.(x) }, f, x) }

basic_apply = →(f, x) { f.(x) }
apply = →(f, x) { map2.(basic_apply, f, x) }

add = →(x, y, z) { x + y + z }

addx = map.(add.curry, Some.new(value: 2))
# ⇒ #<data Some value=Proc>
addy = apply.(addx, Some.new(value: 5))
# ⇒ #<data Some value=Proc>
apply.(addy, Some.new(value: 3))
# ⇒ #<data Some value=10>
```

Map with decider

```
map = ->(fn, decider) {
  Decider.define do
    initial_state fn.(decider.initial_state)

    decide do
      decider.decide(command, state)
    end

    evolve do
      fn.(decider.evolve(state, event))
    end

    terminal? do
      decider.terminal?(state)
    end
  end
}
```

Map2 with decider

```
map = →(fn, dx, dy) {
  Decider.define do
    initial_state fn.(dx.initial_state, dy.initial_state)

    decide do
      dx.decide(command, state) + dy.decide(command, state)
    end

    evolve do
      fn.(
        dx.evolve(state, event),
        dy.evolve(state, event)
      )
    end

    terminal? do
      dx.terminal?(state) && dy.terminal?(state)
    end
  end
}
```

Apply with decider

```
part = map.(fn.curry, decider_x)
part = apply.(part, decider_y)
part = apply.(part, decider_z)
# ⇒ composed decider
```

Decide: run against every decider and return all events

Evolve: apply function with states from all deciders


Why?
Why?
Why?
Oh, that's why.

Few more functions

```ruby
# decider S → decider Hash[id, S]
Decider.many(decider)
decider.many

# decider C1 → decider C2
Decider.lmap_on_command(fn, decider)
decider.lmap_on_command(fn)

# decider S2 → decider S1 → decider S2
Decider.dimap_on_state(fl, fr, decider)
decider.dimap_on_state(fl:, fr:)
```

Few more constructs

```
View.initial_state
# ⇒ state
View.evolve(state, event)
# ⇒ state


Saga.react(event)
# ⇒ [commands]

Process.initial_state
# ⇒ state
Process.evolve(state, event)
# ⇒ state
Process.react(state, event)
# ⇒ [commands]
```

Process + Decider

```
Saga.lmap_on_event
Process.map_on_command

Saga.compose_with_decider(saga, decider)
# ⇒ decider

Process.compose_with_decider(process, decider)
# ⇒ decider
```

Example: split form

```ruby
dx = dx.lmap_on_command(
  ->(command) {
    case command
    in Form(a:, b:)
      Foo.new(a: a, b: b)
    else
      :noop
    end
  }
).lmap_on_state(
  ->(state) { FooState.new(foo: state.foo) }
)
```

```ruby
dy = dy.lmap_on_command(
  ->(command) {
    case command
    in Form(b:, c:)
      Bar.new(b: b, c: c)
    else
      :noop
    end
  }
).lmap_on_state(
  ->(state) { BarState.new(bar: state.bar) }
)
```

Split form

```
decider = Decider.map(
  →(sx, sy) { FormState.new(foo: sx, bar: sy) }.curry,
  dx
).apply(dy)
```

Use cases

Use cases

- form covering multiple aggregates (legacy)

Use cases

- form covering multiple aggregates (legacy)
- public API

Use cases

- form covering multiple aggregates (legacy)
- public API
- webhooks

Use cases

- form covering multiple aggregates (legacy)
- public API
- webhooks
- optional parameters

Use cases

- form covering multiple aggregates (legacy)
- public API
- webhooks
- optional parameters
- low contention (online boardgames!)

Card game

Card game

- `Decider.many(PlayerHandDecider)`

Card game

- `Decider.many(PlayerHandDecider)`
- `TableDecider`

Card game

- `Decider.many(PlayerHandDecider)`
- `TableDecider`
- `DealerDecider`

Card game

- `Decider.many(PlayerHandDecider)`
- `TableDecider`
- `DealerDecider`
- combined with processes for dealing cards, shuffling, determining winner

Card game

- `Decider.many(PlayerHandDecider)`
- `TableDecider`
- `DealerDecider`
- combined with processes for dealing cards, shuffling, determining winner
- single endpoint that can handle the game

Bonus

State can be list

- `initial_state` is empty list
- `decide` takes list of events as state
- `evolve` is redundant - it just appends new events to list

Stronger types

```ruby
require "literal"

State = Literal::Types._Union(Foo, Bar, Baz)

# Just an idea for now
Decider.define(S: State, E: Event, C: Command) do
  # ...
end
```

Totality

A function is total if it is defined for all inputs of its corresponding type, or in other words, if a function returns the output on any possible values of the input types.

https://kowainik.github.io/posts/totality

RBS, LSP

https://joel.drapper.me/p/lsp-driven-design/

Declarative > Imperative

Credits

Jérémie Chassaing

- https://thinkbeforecoding.com/post/2021/12/17/functional-event-sourcing-decider
- https://www.youtube.com/watch?v=bBI-9swoU8c
- https://www.youtube.com/watch?v=72TOhMpEVIA
- https://github.com/thinkbeforecoding/dddeu-2023-deciders
- https://www.youtube.com/watch?v=kgYGMVDHQHs

Ivan Dugalić

- https://fraktalio.com/fmodel/
- https://fraktalio.com/blog/the-template.html
- https://fraktalio.com/blog/types-and-functions.html
- https://fraktalio.com/blog/unmanaged-hazards-exceptions.html
- https://fraktalio.com/blog/side-effects-storing-and-fetching-the-data.html
- https://fraktalio.com/blog/integration-of-event-driven-systems.html
- https://fraktalio.com/blog/context-dependent-declarations.html
- https://fraktalio.com/blog/able-to-be-aggregated.html

Oskar Dudycz

- https://event-driven.io/en/my_journey_from_aggregates/
- https://event-driven.io/en/idempotent_command_handling/
- https://event-driven.io/en/this_is_not_your_uncle_java/
- https://bettersoftwaredesign.pl/podcast/o-implementacji-logiki-biznesowej-z-decider-pattern-z-oskarem-dudyczem/

Yves Goeleven

- linkedin.com/posts/goeleven_many-developers-have-the-wrong-mental-model-activity-7241762871573889024-vWTr
- linkedin.com/posts/goeleven_a-major-mental-difficulty-to-overcome-while-activity-7240675680961597440-8NU0
- linkedin.com/posts/goeleven_as-an-example-for-yesterdays-post-about-activity-7241038041396490240-4aQr
- linkedin.com/posts/goeleven_line-of-business-software-is-nothing-more-activity-7241400480738734081-DaM1
- linkedin.com/posts/goeleven_every-event-model-is-composed-of-transitions-activity-7266057904007401472-7lnH
- linkedin.com/posts/goeleven_i-dont-hate-databases-by-putting-events-activity-7280553431226826754-jbPd

Others

ttps://github.com/MateuszNaKodach/HeroesOfDomainDrivenDesign.EventSourcing.R

ttps://github.com/RailsEventStore/aggregates/tree/master/examples/decider

ttps://dev.to/jakub_zalas/functional-event-sourcing-1ea5

ttps://ismaelcelis.com/posts/decide-evolve-react-pattern-in-ruby/

@jandudulski@ruby.social
@jan.dudulski.pl
jan@dudulski.pl